AFRL-RI-RS-TR-2018-023

ADAPTIVE DECISION MAKING USING PROBABILISTIC PROGRAMMING AND STOCHASTIC OPTIMIZATION

CARNEGIE MELLON UNIVERSITY

JANUARY 2018

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2018-023 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
CHRISTOPHER J. FLYNN
Work Unit Manager

/ S /
JOHN D. MATYJAS
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JAN 2018	FINAL TECHNICAL REPORT	NOV 2016 – AUG 2017

4. TITLE AND SUBTITLE

ADAPTIVE DECISION MAKING USING PROBABILISTIC PROGRAMMING AND STOCHASTIC OPTIMIZATION

5a. CONTRACT NUMBER
FA8750-17-2-0027

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

J. Zico Kolter

5d. PROJECT NUMBER
SAGA

5e. TASK NUMBER
CM

5f. WORK UNIT NUMBER
U1

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2018-023

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This work seeks to understand the connections between learning and decision making under uncertainty. Specifically, we ask that question: when we are going to use learned models within the loop of a larger decision making process, how should we alter the learning procedure or somehow tune the learning to the specific needs of the actual decision making task? To answer this question, we developed a theory of task based model learning, learning models tuned not (just) for predictive accuracy, but to optimize the closed loop performance of a decision making procedure (specifically, those based on stochastic optimization) that uses these models as an intermediate step. Training such models requires that we differentiate through an optimization problem, for which we developed the theory and implementations. On several tasks, we show that such learning substantially outperforms traditional learning processes, where the learning and decision making stages are separate.

15. SUBJECT TERMS

Stochastic programming, machine learning, optimization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**CHRISTOPHER J. FLYNN**
U	U	U	UU	35	19b. TELEPHONE NUMBER (*Include area code*)

Contents

List of Figures

List of Tables

1 Summary

This report details results obtained under the SAGA seedling award, for the project title "Adaptive Decision Making Using Probabilistic Programming and Stochastic Optimization". The work seeks to understand the connections between learning and decision making under uncertainty. Specifically, we ask that question: when we are going to use learned models within the loop of a larger decision making process, how should we alter the learning procedure or somehow tune the learning to the specific needs of the actual decision making task? To answer this question, we develop a theory of task-based model learning, learning models tuned not (just) for predictive accuracy, but to optimize the closed-loop performance of a decision-making procedure (specifically, those based on stochastic optimization) that uses these models as an intermediate step. Training such models requires that we differentiate through an optimization problem, for which we develop the theory and implementations. On several tasks, we show that such learning substantially outperforms traditional learning processes, where the learning and decision-making stages are separate.

2 Introduction

Recent years have seen a growing interest in advancing models for describing and making inferences about stochastic systems. In particular, developments in probabilistic programming have led to a number of high-level frameworks for compactly describing very complex stochastic systems. While a number of probabilistic programming frameworks have emerged, the common theme in these systems has been the ability to express generative probabilistic models using a compact notation. This in turn has enabled researchers to express and reason about much larger systems than was previously possible, without the need for end users to understand the full machinery of the probability methods "beneath the hood." In this sense, they are analogous to classical programming language compilers and interpreters, allowing users to specify desired behaviors at a high level without worrying about the underlying implementation (e.g. the translation to machine code or the conversion to Monte Carlo sampling methods for classical and probabilistic programming methods respectively). For instance, in a general network resilience problem, where the goal is to provide some resource over a constrained network (this could model power grids, communication networks, or traffic networks, for example) a probabilistic program could describe the unknown temporal evolution of demand, along with probabilities that an attacker eliminates a node or additionally constrains the bandwidth along an edge in the graph.

However, probabilistic programming only addresses part of the ultimate goal of decision making under uncertainty. Once we are able to describe and reason about uncertainty in complex domains, the next question that arises is how can we make decisions in these situations in order to achieve some prescribed goal. For example, in the scenarios above, the obvious use of a probabilistic model of supply, demand, and attack probabilities would be to optimize production schedules so as to meet the demand at minimum cost, while ensuring an attack would not bring down the system. These fundamental challenges are the domain of *stochastic optimization*, the task of making decisions in sequential uncertain domains, and under which a wide variety of communities have developed several different classes of approaches. Amongst other, these include: stochastic control [Stengel, 1986], Markov decision processes [Puterman, 2005], stochastic programming [Birge and Louveaux, 2011], stochastic search [Spall, 2003], approximate dynamic programming [Powell, 2011, Bertsekas, 2012]), reinforcement learning [Sutton and Barto, 1998, Szepesvári, 2010], model predictive control [Camacho and Bordons, 2003], and policy search [Mannor et al., 2003, Deisenroth et al., 2011] (to name a few). Recent research has shown that this diverse set of communities can be organized under

a single framework by recognizing that *all* of these problems can be formulated using a common framework that involves searching over policies (see [Powell, 2011][Chapter 5], and [Powell, 2014]). In addition, there are two fundamental strategies for identifying near-optimal policies: policy search, and policies based on lookahead approximations [Powell, 2016].

The key question we address in this study is the following: in the setting where the probabilistic model of the world (i.e., the probabilistic program) is unknown, and where we are ultimately interested in controlling the system to some end, how can we best learn the underlying model? The standard answer to this question from the machine learning community is we can learn our model to maximize the log likelihood of the observed data, or in the case of predicted models, the likelihood of future observations given the past. While this is a natural method, it may fail in cases where the actual underlying system (i.e., the real world) is not well-specified by any model under consideration; in reality, this is the norm rather than the exception for machine learning system.

Instead, the study presented here proposes an alternative answer to the question: *instead of merely learning a model to maximize predictive accuracy, we can learn a model to optimize the performance of the resulting controller when we use the model in a stochastic programming setting.* That is, instead of learning probabilistic models in a task-independent manner, we can learn them in a manner that takes into account the resulting closed-loop performance of the entire system, a strategy known as end-to-end learning, but which is a novel methodology in the stochastic optimization setting.

Although conceptually simple, learning probabilistic models in this manner is not a trivial task, as it involves propagating information through decisions made by the solution to an optimization problem. Thus, the methodology we present here focuses on two key themes required to enable this methodology: *task-based model learning*, and *differentiable optimization*. Briefly, task-based model learning presents the overall paradigm for learning models based upon ultimate task goals, whereas differentiable optimization presents the underlying mathematical framework for actually integrating such goals into a differentiable, deep learning framework.

3 Methods, Assumptions, and Procedures

Here we present the main methodology of our approach, focusing on the two key elements of task-based model learning in stochastic optimization, and differentiable optimization.

3.1 Task-based model learning

Recent advances in AI techniques, notably machine learning and deep learning methods, have shown incredible promise in their ability to injest massive amounts of data and produce models of comensurate complexity that outperform all alternative approaches. These algorithms have been successfully applied to domains such as image recognition, speech recognition, and many others. However, as these big-data methods become more ubiquitous, it is more and more common for the machine learning systems to be integrated "in the loop" of some larger decision making progress. For example, a self-driving car does not merely use deep learning for it's perception, but it subsequently makes decisions based upon this subsystem that hugely effect the overall objective of the vehicle (to safely drive a passenger from point A to point B). Similarly for a domain that we will consider as a primary application of our methodology, that of determining how best to schedule the electrical generation in the power grid so as to maximize the efficiency, reliability, and security of the power grid, it is standard practice to use complex forecasting models to predict how the grid and it's occupants are likely to behave. In these domains, there is more than just big data, there are "big

decisions", large-scale decision making and optimization problems that need to be addressed using the data themselves.

Yet a common them in these and many big-data machine learning systems is that the underlying models are learned *independently* of their ultimate use in a task-based manner. We typically train such models to minimize some "simple" notion of loss or other error metric, often without any consideration of the ultimate decision-making process for which the model will be employed. And when the ultimate "task loss" (the task objective that we are ultimately trying to optimize) has substantially asymmetric costs — i.e., the costs of overgeneration of electricity (which wastes resources) versus undergeneration (which can cause blackouts) — then ignoring the ultimate task can lead to models that perform substantially worse that necessary on the complete decision-making process. An alternative approach, of course, is to use a *fully* black-box model, such as those employed in some deep reinforcement learning settings, that attempt to directly solve the entire end-to-end decision-making problem purely from data; but this approach is equally fraught with a lack of transparency or interpretability in the decision-making process, typically a need for much more data, and generally seems to be unlikely to be deployed soon in the most critical decision-making tasks.

In this work, we propose an alternative approach that lies in between these two extremes. In particular, we develop an approach that remains model-based: we still use a (potentially deep) machine learning algorithm to estimate a model from large amounts of data, which we then use in support of a decision-making system. But the key element here is that we train the model *specifically to minimize the ultimate task loss*; that is, we train the model itself in an end-to-end fashion to minimize the ultimate loss that we care about. The main goal of this proposed work is to develop the algorithms and analysis to enable this method to be scaled to the size of data that current deep learning approach apply to, to explicitly enable big decision making from huge amounts a data. We propose to train a probabilistic model not (solely) for predictive accuracy, but so that–when it is later used within the loop of a stochastic programming procedure–it produces solutions that minimize the ultimate task-based loss. This formulation may seem somewhat counterintuitive, given that a "perfect" predictive model would of course also be the optimal model to use within a stochastic programming framework. However, the reality that all models *do* make errors illustrates that we should indeed look to a final task-based objective to determine the proper error tradeoffs within a machine learning setting. This work proposes one way to evaluate task-based tradeoffs in a fully automated fashion, by computing derivatives through the solution to the stochastic programming problem in a manner that can improve the underlying model.

We begin by presenting background material and related work in areas spanning stochastic programming, end-to-end training, optimizing alternative loss functions, and the classic generative/discriminative tradeoff in machine learning. We then describe our approach within the formal context of stochastic programming, and give a generic method for propagating task loss through these problems in a manner that can update the models. We report on two experimental evaluations of the proposed approach, on a classical inventory stock problem and on a real-world electrical grid scheduling task. In both cases we show that the proposed approach outperforms traditional modeling and purely black-box policy optimization approaches.

3.1.1 Background and Related Work

Stochastic programming Stochastic programming is a method for making decisions under uncertainty by modeling or optimizing objectives governed by a random process. It has applications in many domains such as energy [Wallace and Fleten, 2003], finance [Ziemba and Vickson, 2006], and manufacturing [Buzacott and Shanthikumar, 1993], where the underlying probability distributions

are either known or can be estimated. Common considerations include how to best model or approximate the underlying random variable, how to solve the resulting optimization problem, and how to then assess the quality of the resulting (approximate) solution [Shapiro and Philpott, 2007].

In cases where the underlying probability distribution is known but the objective cannot be solved analytically, it is common to use Monte Carlo sample average approximation methods, which draw multiple i.i.d. samples from the underlying probability distribution and then use deterministic optimization methods [Linderoth et al., 2006]. In cases where the underlying distribution is not known, it is common to learn or estimate some model from observed samples [Rockafellar and Wets, 1991].

End-to-end training Recent years have seen a dramatic increase in the number of systems building on so-called "end-to-end" learning. Generally speaking, this term refers to systems where the end goal of the machine learning process is directly predicted from the raw inputs [e.g. LeCun et al., 2005, Thomas et al., 2006]. In the context of deep learning systems, the term now traditionally refers to architectures where there is no, for example, explicit encoding of hand-tuned features on the data, but the system directly predicts what the image, text, etc. is from the raw inputs [Wang et al., 2011, He et al., 2016, Wang et al., 2012, Graves and Jaitly, 2014, Amodei et al., 2015]. The context in which we use the term end-to-end is similar, but slightly more in line with its older usage: instead of (just) attempting to learn an output (with known and typically straightforward loss functions), we are specifically attempting to learn a model based upon an end-to-end *task* that the user is ultimately trying to accomplish. We feel that this concept–of describing the entire closed-loop performance of the system as evaluated on the real task at hand–is beneficial to add to the notion of end-to-end learning.

Also highly related to our work are recent efforts in end-to-end policy learning [Levine et al., 2016], using value iteration effectively as an optimization procedure in similar networks [Tamar et al., 2016], and multi-objective optimization [Harada et al., 2006, Van Moffaert and Nowé, 2014, Mossalam et al., 2016, Wiering et al., 2014]. These lines of work fit more with the "pure" end-to-end approach we discuss below (where models are eschewed for pure function approximation methods), but conceptually the approaches have similar motivations in modifying typically-optimized policies to address some task(s) directly. Of course, the actual methodological approaches are quite different, given our specific focus on stochastic programming as the black box of interest in our setting.

Optimizing alternative loss functions There has been a great deal of work in recent years on using machine learning procedures to optimize different loss criteria than those "naturally" optimized by the algorithm. For example, Stoyanov et al. [2011] and Hazan et al. [2010] propose methods for optimizing loss criteria in structured prediction that are *different* from the inference procedure of the prediction algorithm; this work has also recently been extended to deep networks [Song et al., 2016]. Recent work has also explored using auxiliary prediction losses to simultaneously satisfy multiple objectives [Jaderberg et al., 2016].

The work that we have found in the literature that most closely resembles our approach is the work of Bengio [1997], which used a neural network model for predicting financial prices, and then optimized the model based upon returns obtained via a hedging strategy that employed it. We view this approach–of both using a model but then tuning that model to adapt to a (differentiable) procedure–as a philosophical predecessor to our own work. Whereas Bengio [1997]'s work used a hand-crafted (but differentiable) algorithm to approximately attain some objective given a predictive model, our approach is tightly coupled to stochastic programming, where the explicit objective is to *attempt* to optimize the desired task cost via an exact optimization routine,

but given underlying randomness. The notions of stochasticity are thus naturally quite different in our own work, but we do hope that our work can bring back the original idea of task-based model learning. (Despite the original paper being nearly 20 years old, virtually all follow-on work has focused on the financial application, and not on what we feel is the core idea of using a surrogate model within a task-driven optimization procedure.) We note that Bansal et al. [2017] also recently employed a similar concept to learn dynamics models that maximize control performance, but in the context of Bayesian optimization.

3.1.2 End-to-end model learning in stochastic programming

We first formally define the stochastic modeling and optimization problems with which we are concerned. Let $(x \in \mathcal{X}, y \in \mathcal{Y}) \sim \mathcal{D}$ denote standard input, output pairs drawn from some (real, unknown) distribution $\mathcal{D}$. We also consider actions $z \in \mathcal{Z}$ that incur some expected loss $L_{\mathcal{D}}(z) = E_{x,y\sim\mathcal{D}}[f(x, y, z)]$. For instance, a power systems operator may try to allocate power generators z given given past electricity demand x and future electricity demand y; this allocation's loss is the over- or under-generation penalties incurred given future demand instantiations.

If we knew $\mathcal{D}$, then we could select optimal actions $z^\star_{\mathcal{D}} = \text{argmin}_z\, L_{\mathcal{D}}(z)$. However, in practice, the true distribution $\mathcal{D}$ is unknown. In this section, we are interested in modeling the conditional distribution $y|x$ using some parameterized model $p(y|x;\theta)$ in order to minimize the real-world cost of the policy implied by this parameterization. Specifically, we find some parameters θ to parameterize $p(y|x;\theta)$ (as in the standard statistical setting) and then determine (via stochastic optimization) optimal actions $z^\star(x;\theta)$ that correspond to our observed input x and the specific choice of parameters θ in our probabilistic model. Upon observing the costs of these actions $z^\star(x;\theta)$ relative to true instantiations of x and y, we update our parameterized model $p(y|x;\theta)$ accordingly, calculate the resultant new $z^\star(x;\theta)$, and repeat. The goal is to find parameters θ such that the corresponding policy $z^\star(x;\theta)$ optimizes loss under the *true* joint distribution of x and y.

Explicitly, we wish to choose θ to minimize the *task loss* $L(\theta)$ in the context of $x, y \sim \mathcal{D}$, i.e.

$$\underset{\theta}{\text{minimize}} \quad L(\theta) = \mathbf{E}_{x,y\sim\mathcal{D}}[f(x, y, z^\star(x;\theta))]. \tag{1}$$

Since in reality we do not know the distribution $\mathcal{D}$, we obtain $z^\star(x;\theta)$ via a proxy stochastic optimization problem for a fixed instantiation of parameters θ, i.e.

$$z^\star(x;\theta) = \underset{z}{\text{argmin}} \quad \mathbf{E}_{y\sim p(y|x;\theta)}[f(x, y, z)]. \tag{2}$$

The above setting specifies $z^\star(x;\theta)$ using a simple (unconstrained) stochastic program, but in reality our decision must be made subject to both probabilistic and deterministic constraints. We therefore consider more general decisions produced through a generic stochastic programming problem[1]

$$\begin{aligned} z^\star(x;\theta) = \underset{z}{\text{argmin}} \;\; & \mathbf{E}_{y\sim p(y|x;\theta)}[f(x, y, z)] \\ \text{subject to} \;\; & \mathbf{E}_{y\sim p(y|x;\theta)}[g_i(x, y, z)] \leq 0, \quad i = 1, \ldots, n_{ineq} \\ & h_i(z) = 0, \quad i = 1, \ldots, n_{eq}. \end{aligned} \tag{3}$$

In this setting, the full task loss is more complex, since it captures both the expected cost and any deviations from the constraints. We can write this, for instance, as

$$L(\theta) = \mathbf{E}_{x,y\sim\mathcal{D}}[f(x, y, z^\star(x;\theta))] + \sum_{i=1}^{n_{ineq}} I\{\mathbf{E}_{x,y\sim\mathcal{D}}[g_i(x, y, z^\star(x;\theta))] \leq 0\} + \sum_{i=1}^{n_{eq}} I\{h_i(z^\star(x;\theta)) = 0\} \tag{4}$$

[1] It is standard to presume in stochastic programming that equality constraints depend only on decision variables (not random variables), as non-trivial random equality constraints are typically not possible to satisfy.

(where $I(\cdot)$ is the indicator function that is zero when its constraints are satisfied and infinite otherwise). However, the basic intuition behind our approach remains the same for both the constrained and unconstrained cases: in both settings, we attempt to learn parameters of a probabilistic model not to produce strictly "accurate" predictions, but such that *when we use the resultant model within a stochastic programming setting, the resulting decisions perform well under the true distribution.*

However, actually solving this problem requires that we differentiate through the "argmin" operator $z^\star(x;\theta)$ of the stochastic programming problem; while this differentiation is not possible for all classes of optimization problems (the argmin operator may be discontinuous), in many practical cases (including cases where the function and constraints are strongly convex), as we will show shortly, we can indeed efficiently compute these gradients even in the context of constrained optimization.

3.1.3 Discussion and alternative approaches

We highlight our approach in contrast to two alternative existing methods: traditional model learning and model-free black-box policy optimization. In traditional ML approaches, it is common to use θ to minimize the (conditional) log-likelihood of observed data under the model $p(y|x;\theta)$. This method corresponds to approximately solving the optimization problem

$$\underset{\theta}{\text{minimize}} \quad \mathbf{E}_{x,y\sim\mathcal{D}}\left[-\log p(y|x;\theta)\right]. \tag{5}$$

If we then need to use the conditional distribution $y|x$ to determine actions z within some later optimization setting, we commonly use the predictive model obtained from (5) directly. This approach has obvious advantages, in that the model-learning phase is well-justified independent of any future use in a task. However, it is also prone to poor performance in the common setting where the true distribution $y|x$ cannot be represented within the class of distribution parameterized by θ. Conceptually, the log-likelihood objective *implicitly* trades off between model error in different regions of the input/output space, but does so in a manner largely opaque to the modeler, and may ultimately *not* employ the correct tradeoffs for a given task.

In contrast, there is an alternative approach to solving (1) that we describe as the model-free "black-box" policy optimization approach. Here, we forgo learning any model at all of the random variable y. Instead, we attempt to learn a policy mapping directly from inputs x to actions $z^\star(x;\bar{\theta})$ that minimize the loss $L(\bar{\theta})$ presented in (4) (where here $\bar{\theta}$ defines the form of the policy itself, not a predictive model). While such model-free methods can perform well in many settings, they are often very data inefficient, as the policy class must have enough representational power to describe sufficiently complex policies without recourse to any underlying model.[2]

[2]This distinction is roughly analogous to the policy search vs. model-based settings in reinforcement learning. However, for the purposes of this paper, we consider much simpler stochastic programs without the multiple rounds that occur in RL, and the extension of these techniques to a full RL setting remains as future work.

Algorithm 1 Task Loss Optimization

Input: Example $x, y \sim D$.
initialize θ *// some initial parameterization*
for $t = 1, \ldots, T$ **do**
  compute $z^\star(x;\theta)$ via Equation (3)
  initialize α_t *// step size*
  // step in violated constraint or objective
  if $\exists i$ s.t. $g_i(x, y, z^\star(x;\theta)) > 0$ **then**
    update $\theta \leftarrow \theta - \alpha_t \nabla_\theta g_i(x, y, z^\star(x;\theta))$
  else
    update $\theta \leftarrow \theta - \alpha_t \nabla_\theta f(x, y, z^\star(x;\theta))$
  end if
end for

Our approach offers an intermediate setting, where we *do* still use a surrogate model to determine an optimal decision $z^\star(x;\theta)$, yet we adapt this model based on the task loss instead of any model prediction accuracy. In practice, we typically want to minimize some weighted combination of log-likelihood *and* task loss, which can be easily accomplished given our approach.

3.1.4 Optimizing task loss

To solve the generic optimization problem (1), we can in principle adopt a straightforward (constrained) stochastic gradient approach, as detailed in Algorithm 1. At each iteration, we solve the proxy stochastic programming problem (3) to obtain $z^\star(x, \theta)$, using the distribution defined by our current values of θ. Then, we compute the true loss $L(\theta)$ using the observed value of y. If any of the inequality constraints g_i in $L(\theta)$ are violated, we take a gradient step in the violated constraint; otherwise, we take a gradient step in the optimization objective f. We note that this approach requires training in mini-batches to determine whether probabilistic constraints are satisfied. Alternatively, because even the g_i constraints are probabilistic, it is common in practice to simply move a weighted version of these constraints to the objective: i.e., we modify the objective by adding some appropriate penalty times the positive part of the function, $\lambda g_i(x, y, z)_+$, for some $\lambda > 0$. In practice, this has the effect of taking gradient steps jointly in all the violated constraints and the objective in the case that one or more inequality constraints are violated, often resulting in faster convergence. Note that we need only move stochastic constraints into the objective; deterministic constraints on the policy itself will always be satisfied by the optimizer, as they are independent of the model.

3.2 Differentiable Optimization

In this section, we consider how to treat exact, constrained optimization as an individual layer within a deep learning architecture. Unlike traditional feedforward networks, where the output of each layer is a relatively simple (though non-linear) function of the previous layer, our optimization framework allows for individual layers to capture much richer behavior, expressing complex operations that in total can reduce the overall depth of the network while preserving richness of representation. Specifically, we build a framework where the output of the $i+1$th layer in a network is the *solution* to a constrained optimization problem based upon previous layers. This framework naturally encompasses a wide variety of inference problems expressed within a neural network, allowing for the potential of much richer end-to-end training for complex tasks that require such inference procedures.

Concretely, we specifically consider the task of solving small quadratic programs as individual layers. These optimization problems are well-suited to capturing interesting behavior and can be efficiently solved with GPUs. Specifically, we consider layers of the form

$$\begin{aligned} z_{i+1} = \underset{z}{\text{argmin}} \;\; & \frac{1}{2} z^T Q(z_i) z + q(z_i)^T z \\ \text{subject to} \;\; & A(z_i) z = b(z_i) \\ & G(z_i) z \leq h(z_i) \end{aligned} \tag{6}$$

where z is the optimization variable, $Q(z_i)$, $q(z_i)$, $A(z_i)$, $b(z_i)$, $G(z_i)$, and $h(z_i)$ are parameters of the optimization problem. As the notation suggests, these parameters can depend in any differentiable way on the previous layer z_i, and which can eventually be optimized just like any other weights in a neural network. These layers can be learned by taking the gradients of some loss function with respect to the parameters. We derive the gradients of (6) by taking matrix differentials of the KKT conditions of the optimization problem at its solution.

In order to the make the approach practical for larger networks, we develop a custom solver which can simultaneously solve multiple small QPs in batch form. We do so by developing a custom primal-dual interior point method tailored specifically to dense batch operations on a GPU. In total, the solver can solve batches of quadratic programs over 100 times faster than existing highly tuned quadratic programming solvers such as Gurobi and CPLEX. One crucial algorithmic insight in the solver is that by using a specific factorization of the primal-dual interior point update, we can obtain a backward pass over the optimization layer virtually "for free" (i.e., requiring no additional factorization once the optimization problem itself has been solved). Together, these innovations enable parameterized optimization problems to be inserted within the architecture of existing deep networks.

We begin by highlighting background and related work, and then present our optimization layer itself. Using matrix differentials we derive rules for computing all the necessary backpropagation updates. We then detail our specific solver for these quadratic programs, based upon a state-of-the-art primal-dual interior point method, and highlight the novel elements as they apply to our formulation, such as the aforementioned fact that we can compute backpropagation at very little additional cost. We then provide experimental results that demonstrate the capabilities of the architecture, highlighting potential tasks that these architectures can solve, and illustrating improvements upon existing approaches.

3.2.1 Background and related work

Optimization plays a key role in modeling complex phenomena and providing concrete decision making processes in sophisticated environments. A full treatment of optimization applications in beyond our scope, [Boyd and Vandenberghe, 2004], but these methods have bound applicability in control frameworks [Morari and Lee, 1999, Sastry and Bodson, 2011]; numerous statistical and mathematical formalisms [Sra et al., 2012], and physical simulation problems like rigid body dynamics [Lötstedt, 1984]. Generally speaking, our work is a step towards learning optimization problems behind real-world processes from data that can be learned end-to-end rather than requiring human specification and intervention.

In the machine learning setting, a wide array of applications consider optimization as a means to perform inference in learning. Among many other applications, these architectures are well-studied for generic classification and structured prediction tasks [Goodfellow et al., 2013, Stoyanov et al., 2011, Brakel et al., 2013, LeCun et al., 2006, Belanger and McCallum, 2016, Belanger et al.,

2017, Amos et al., 2016]; in vision for tasks such as denoising [Tappen et al., 2007, Schmidt and Roth, 2014]; and [Metz et al., 2016] uses unrolled optimization within a network to stabilize the convergence of generative adversarial networks [Goodfellow et al., 2014]. Indeed, the general idea of solving restricted classes of optimization problem using neural networks goes back many decades [Kennedy and Chua, 1988, Lillo et al., 1993], but has seen a number of advances in recent years. These models are often trained by one of the following four methods.

Energy-based learning methods These methods can be used for tasks like (structured) prediction where the training method shapes the energy function to be low around the observed data manifold and high elsewhere [LeCun et al., 2006]. In recent years, there has been a strong push to further incorporate structured prediction methods like conditional random fields as the "last layer" of a deep network architecture [Peng et al., 2009, Zheng et al., 2015, Chen et al., 2015] as well as in deeper energy-based architectures [Belanger and McCallum, 2016, Belanger et al., 2017, Amos et al., 2016]. Learning in this context requires observed data, which isn't present in some of the contexts we consider in this work, and also may suffer from instability issues when combined with deep energy-based architectures as observed in Belanger and McCallum [2016], Belanger et al. [2017], Amos et al. [2016].

Analytically If an analytic solution to the argmin can be found, such as in an unconstrained quadratic minimization, the gradients can often be computed analytically. This is done in [Tappen et al., 2007, Schmidt and Roth, 2014]. We cannot use these methods for the constrained optimization problems we consider in this work because there are no known analytic solutions.

Unrolling The argmin operation over an unconstrained objective can be approximated by a first-order gradient-based method and unrolled. These architectures typically introduce an optimization procedure such as gradient descent into the inference procedure. This is done in [Domke, 2012, Amos et al., 2016, Belanger et al., 2017, Metz et al., 2016, Goodfellow et al., 2013, Stoyanov et al., 2011, Brakel et al., 2013], The optimization procedure is unrolled automatically or manually [Domke, 2012] to obtain derivatives during training that incorporate the effects of these in-the-loop optimization procedures. However, unrolling the computation of a method like gradient descent typically requires a substantially larger network, and adds substantially to the computational complexity of the network.

In all of these existing cases, the optimization problem is unconstrained and unrolling gradient descent is often easy to do. When constraints are added to the optimization problem, iterative algorithms often use a projection operator that may be difficult to unroll through. In our work, we do **not** unroll an optimization procedure but instead use argmin differentiation as described in the next section.

Argmin differentiation Most closely related to our own work, there have been several papers that propose some form of differentiation through argmin operators. These techniques also come up in bi-level optimization [Gould et al., 2016, Kunisch and Pock, 2013] and sensitivity analysis [Bertsekas, 1999, Fiacco and Ishizuka, 1990, Bonnans and Shapiro, 2013]. In the case of Gould et al. [2016], the authors describe general techniques for differentiation through optimization problems, but only describe the case of exact equality constraints rather than both equality and inequality constraints (in the case inequality constraints, they add these via a barrier function). Amos et al. [2016] considers argmin differentiation within the context of a specific optimization problem (the bundle method) but does not consider a general setting. Johnson et al. [2016] performs implicit

differentiation on (multi-)convex objectives with coordinate subspace constraints, but don't consider inequality constraints and don't consider in detail general linear equality constraints. Their optimization problem is only in the final layer of a variational inference network while we propose to insert optimization problems anywhere in the network. Therefore a special case of OptNet layers (with no inequality constraints) has a natural interpretation in terms of Gaussian inference, and so Gaussian graphical models (or CRF ideas more generally) provide tools for making the computation more efficient and interpreting or constraining its structure. Similarly, the older work of [Mairal et al., 2012] considered argmin differentiation for a LASSO problem, deriving specific rules for this case, and presenting an efficient algorithm based upon our ability to solve the LASSO problem efficiently.

We use implicit differentiation [Dontchev and Rockafellar, 2009, Griewank and Walther, 2008] and techniques from matrix differential calculus [Magnus and Neudecker, 1988] to derive the gradients from the KKT matrix of the problem we are interested in. A notable different from other work within ML that we are aware of, is that we analytically differentiate through inequality as well as just equality constraints, but differentiating the complementarity conditions; this differs from e.g., Gould et al. [2016] where they instead approximately convert the problem to an unconstrained one via a barrier method. We have also developed methods to make this approach practical and reasonably scalable within the context of deep architectures.

3.2.2 OptNet: solving optimization within a neural network

Although in the most general form, an OptNet layer can be any optimization problem, in this work we will study OptNet layers defined by a quadratic program

$$\begin{aligned} \underset{z}{\text{minimize}} \;\; & \frac{1}{2} z^T Q z + q^T z \\ \text{subject to} \;\; & Az = b, \; Gz \leq h \end{aligned} \tag{7}$$

where $z \in \mathbb{R}^n$ is our optimization variable $Q \in \mathbb{R}^{n \times n} \succeq 0$ (a positive semidefinite matrix), $q \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $G \in \mathbb{R}^{p \times n}$ and $h \in \mathbb{R}^p$ are problem data, and leaving out the dependence on the previous layer z_i for notational convenience. As is well-known, these problems can be solved in polynomial time using a variety of methods; if one desires exact (to numerical precision) solutions to these problems, then primal-dual interior point methods, as we will use in a later section, are the current state of the art in solution methods. In the neural network setting, the *optimal solution* (or more generally, a *subset of the optimal solution*) of this optimization problems becomes the output of our layer, denoted z_{i+1}, and any of the problem data Q, q, A, b, G, h can depend on the value of the previous layer z_i. The forward pass in our OptNet architecture thus involves simply setting up and finding the solution to this optimization problem.

Training deep architectures, however, requires that we not just have a forward pass in our network but also a backward pass. This requires that we compute the derivative of the solution to the QP with respect to its input parameters, a general topic we topic we discussed previously. To obtain these derivatives, we differentiate the KKT conditions (sufficient and necessary condition for optimality) of (6) at a solution to the problem using techniques from matrix differential calculus [Magnus and Neudecker, 1988]. Our analysis here can be extended to more general convex optimization problems.

The Lagrangian of (6) is given by

$$L(z, \nu, \lambda) = \frac{1}{2} z^T Q z + q^T z + \nu^T (Az - b) + \lambda^T (Gz - h) \tag{8}$$

where ν are the dual variables on the equality constraints and $\lambda \geq 0$ are the dual variables on the inequality constraints. The KKT conditions for stationarity, primal feasibility, and complementary slackness are

$$\begin{aligned} Qz^\star + q + A^T\nu^\star + G^T\lambda^\star &= 0 \\ Az^\star - b &= 0 \\ D(\lambda^\star)(Gz^\star - h) &= 0, \end{aligned} \tag{9}$$

where $D(\cdot)$ creates a diagonal matrix from a vector. Taking the differentials of these conditions gives the equations

$$\begin{aligned} \mathsf{d}Qz^\star + Q\mathsf{d}z + \mathsf{d}q + \mathsf{d}A^T\nu^\star + \\ A^T\mathsf{d}\nu + \mathsf{d}G^T\lambda^\star + G^T\mathsf{d}\lambda = 0 \\ \mathsf{d}Az^\star + A\mathsf{d}z - \mathsf{d}b = 0 \\ D(Gz^\star - h)\mathsf{d}\lambda + D(\lambda^\star)(\mathsf{d}Gz^\star + G\mathsf{d}z - \mathsf{d}h) = 0 \end{aligned} \tag{10}$$

or written more compactly in matrix form

$$\begin{aligned} &\begin{bmatrix} Q & G^T & A^T \\ D(\lambda^\star)G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathsf{d}z \\ \mathsf{d}\lambda \\ \mathsf{d}\nu \end{bmatrix} = \\ &\qquad \begin{bmatrix} -\mathsf{d}Qz^\star - \mathsf{d}q - \mathsf{d}G^T\lambda^\star - \mathsf{d}A^T\nu^\star \\ -D(\lambda^\star)\mathsf{d}Gz^\star + D(\lambda^\star)\mathsf{d}h \\ -\mathsf{d}Az^\star + \mathsf{d}b \end{bmatrix}. \end{aligned} \tag{11}$$

Using these equations, we can form the Jacobians of $z^\star$ (or $\lambda^\star$ and $\nu^\star$, though we don't consider this case here), with respect to any of the data parameters. For example, if we wished to compute the Jacobian $\frac{\partial z^\star}{\partial b} \in \mathbb{R}^{n \times m}$, we would simply substitute $\mathsf{d}b = I$ (and set all other differential terms in the right hand side to zero), solve the equation, and the resulting value of $\mathsf{d}z$ would be the desired Jacobian.

In the backpropagation algorithm, however, we never want to explicitly form the actual Jacobian matrices, but rather want to form the left matrix-vector product with some previous backward pass vector $\frac{\partial \ell}{\partial z^\star} \in \mathbb{R}^n$, i.e., $\frac{\partial \ell}{\partial z^\star}\frac{\partial z^\star}{\partial b}$. We can do this efficiently by noting the solution for the $(\mathsf{d}z, \mathsf{d}\lambda, \mathsf{d}\nu)$ involves multiplying the *inverse* of the left-hand-side matrix in (11) by some right hand side. Thus, if we multiply the backward pass vector by the transpose of the differential matrix

$$\begin{bmatrix} d_z \\ d_\lambda \\ d_\nu \end{bmatrix} = \begin{bmatrix} Q & G^T D(\lambda^\star) & A^T \\ G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \left(\frac{\partial \ell}{\partial z^\star}\right)^T \\ 0 \\ 0 \end{bmatrix} \tag{12}$$

then the relevant gradients with respect to all the QP parameters can be given by

$$\begin{aligned} \frac{\partial \ell}{\partial q} &= d_z & \frac{\partial \ell}{\partial b} &= -d_\nu \\ \frac{\partial \ell}{\partial h} &= -D(\lambda^\star)d_\lambda & \frac{\partial \ell}{\partial Q} &= \frac{1}{2}(d_z z^T + z d_z^T) \\ \frac{\partial \ell}{\partial A} &= d_\nu z^T + \nu d_z^T & \frac{\partial \ell}{\partial G} &= D(\lambda^\star)(d_\lambda z^T + \lambda d_z^T) \end{aligned} \tag{13}$$

where as in standard backpropagation, all these terms are at most the size of the parameter matrices. As we will see in the next section, the solution to an interior point method in fact already provides a factorization we can use to compute these gradient efficiently.

3.2.3 An efficient batched QP solver

Deep networks are typically trained in mini-batches to take advantage of efficient data-parallel GPU operations. Without mini-batching on the GPU, many modern deep learning architectures become intractable for all practical purposes. However, today's state-of-the-art QP solvers like Gurobi and CPLEX do not have the capability of solving multiple optimization problems on the GPU in parallel across the entire minibatch. This makes larger OptNet layers become quickly intractable compared to a fully-connected layer with the same number of parameters.

To overcome this performance bottleneck in our quadratic program layers, we have implemented a GPU-based primal-dual interior point method (PDIPM) based on [Mattingley and Boyd, 2012] that solves a batch of quadratic programs, and which provides the necessary gradients needed to train these in an end-to-end fashion. Our performance experiments in Section 4.2.1 shows that our solver is significantly faster than the standard non-batch solvers Gurobi and CPLEX.

Following the method of [Mattingley and Boyd, 2012], our solver introduces slack variables on the inequality constraints and iteratively minimizes the residuals from the KKT conditions over the primal variable $z \in \mathbb{R}^n$, slack variable $s \in \mathbb{R}^p$, and dual variables $\nu \in \mathbb{R}^m$ associated with the equality constraints and $\lambda \in \mathbb{R}^p$ associated with the inequality constraints. Each iteration computes the affine scaling directions by solving

$$K \begin{bmatrix} \Delta z^{\text{aff}} \\ \Delta s^{\text{aff}} \\ \Delta \lambda^{\text{aff}} \\ \Delta \nu^{\text{aff}} \end{bmatrix} = \begin{bmatrix} -(A^T \nu + G^T \lambda + Qz + q) \\ -S\lambda \\ -(Gz + s - h) \\ -(Az - b) \end{bmatrix} \tag{14}$$

where

$$K = \begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & D(\lambda) & D(s) & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix},$$

then centering-plus-corrector directions by solving

$$K \begin{bmatrix} \Delta z^{\text{cc}} \\ \Delta s^{\text{cc}} \\ \Delta \lambda^{\text{cc}} \\ \Delta \nu^{\text{cc}} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma\mu 1 - D(\Delta s^{\text{aff}})\Delta\lambda^{\text{aff}} \\ 0 \\ 0 \end{bmatrix}, \tag{15}$$

where $\mu = s^T\lambda/p$ and σ is defined in [Mattingley and Boyd, 2012]. Each variable v is updated with $\Delta v = \Delta v^{\text{aff}} + \Delta v^{\text{cc}}$ using an appropriate step size. We actually solve a symmetrized version of the KKT conditions, obtained by scaling the second row block by $D(1/s)$. We analytically decompose these systems into smaller symmetric systems and pre-factorize portions of them that don't change (i.e. that don't involve $D(\lambda/s)$ between iterations). We have implemented a batched version of this method with the PyTorch library[3] and have released it as an open source library at `https://github.com/locuslab/qpth`. It uses a custom CUBLAS extension that provides an interface to solve multiple matrix factorizations and solves in parallel, and which provides the necessary backpropagation gradients for their use in an end-to-end learning system.

[3] `https://pytorch.org`

3.2.4 Efficiently computing gradients

A key point of the particular form of primal-dual interior point method that we employ is that it is possible to compute the backward pass gradients "for free" after solving the original QP, without an additional matrix factorization or solve. Specifically, at each iteration in the primal-dual interior point, we are computing an LU decomposition of the matrix K_{sym}.[4] This matrix is essentially a symmetrized version of the matrix needed for computing the backpropagated gradients, and we can similarly compute the $d_{z,\lambda,\nu}$ terms by solving the linear system

$$K_{\text{sym}} \begin{bmatrix} d_z \\ d_s \\ \tilde{d}_\lambda \\ d_\nu \end{bmatrix} = \begin{bmatrix} \left(-\frac{\partial \ell}{\partial z_{i+1}}\right)^T \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{16}$$

where $\tilde{d}_\lambda = D(\lambda^\star) d_\lambda$ for d_λ as defined in (12). Thus, all the backward pass gradients can be computed using the factored KKT matrix at the solution. Crucially, because the bottleneck of solving this linear system is computing the factorization of the KKT matrix (cubic time as opposed to the quadratic time for solving via backsubstitution once the factorization is computed), the additional time requirements for computing all the necessary gradients in the backward pass is virtually nonexistent compared with the time of computing the solution. To the best of our knowledge, this is the first time that this fact has been exploited in the context of learning end-to-end systems.

3.2.5 Properties and representational power

In this section we briefly highlight some of the mathematical properties of OptNet layers. The proofs here are straightforward, and are mostly based upon well-known results in convex analysis, so are deferred to the appendix. The first result simply highlights that (because the solution of strictly convex QPs is continuous), that OptNet layers are subdifferentiable everywhere, and differentiable at all but a measure-zero set of points.

Theorem 1. *Let $z^\star(\theta)$ be the output of an OptNet layer, where $\theta = \{Q, p, A, b, G, h\}$. Assuming $Q \succ 0$ and that A has full row rank, then $z^\star(\theta)$ is subdifferentiable everywhere: $\partial z^\star(\theta) \neq \varnothing$, where $\partial z^\star(\theta)$ denotes the Clarke generalized subdifferential [Clarke, 1975] (an extension of the subgradient to non-convex functions), and has a single unique element (the Jacobian) for all but a measure zero set of points θ.*

Proof. The fact that an OptNet layer is subdifferentiable from strictly convex QPs ($Q \succ 0$) follows directly from the well-known result that the solution of a strictly convex QP is continuous (though not everywhere differentiable). Our proof essentially just boils down to showing this fact, though we do so by explicitly showing that there *is* a unique solution to the Jacobian equations (11) that we presented earlier, except on a measure zero set. This measure zero set consists of QPs with degenerate solutions, points where inequality constraints can hold with equality yet also have zero-valued dual variables. For simplicity we assume that A has full row rank, but this can be

[4] We actually perform an LU decomposition of a certain subset of the matrix formed by eliminating variables to create only a $p \times p$ matrix (the number of inequality constraints) that needs to be factor during each iteration of the primal-dual algorithm, and one $m \times m$ and one $n \times n$ matrix once at the start of the primal-dual algorithm, though we omit the detail here. We also use an LU decomposition as this routine is provided in batch form by CUBLAS, but could potentially use a (faster) Cholesky factorization if and when the appropriate functionality is added to CUBLAS).

From the complementarity condition, we have that at a primal dual solution $(z^\star, \lambda^\star, \nu^\star)$

$$\begin{aligned} (Gz^\star - h)_i < 0 &\rightarrow \lambda_i^\star = 0 \\ \lambda_i^\star > 0 &\rightarrow (Gz^\star - h)_i = 0 \end{aligned} \tag{17}$$

(i.e., we cannot have both these terms non-zero).

First we consider the (typical) case where exactly one of $(Gz^\star - h)_i$ and $\lambda_i^\star$ is zero. Then the KKT differential matrix

$$\begin{bmatrix} Q & G^T & A^T \\ D(\lambda^\star)G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix} \tag{18}$$

(the left hand side of (11)) is non-singular. To see this, note that if we let $\mathcal{I}$ be the set where $\lambda_i^\star > 0$, then the matrix

$$\begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & D(Gz^\star - h)_{\mathcal{I}} & 0 \\ A & 0 & 0 \end{bmatrix} = \begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & 0 & 0 \\ A & 0 & 0 \end{bmatrix} \tag{19}$$

is non-singular (scaling the second block by $D(\lambda^\star)^{-1}$ gives a standard KKT system [Boyd and Vandenberghe, 2004, Section 10.4], which is nonsingular for invertible Q and $[G_{\mathcal{I}}^T \quad A^T]$ with full column rank, which must hold due to our condition on A and the fact that there must be less than n total tight constraints at the solution. Also note that for any $i \notin \mathcal{I}$, only the $D(Gz^\star - h)_{ii}$ term in non-zero for the entire row in the second block of the matrix. Thus, if we want to solve the system

$$\begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & D(Gz^\star - h)_{\mathcal{I}} & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda \\ \nu \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{20}$$

we simply first set $\lambda_i = b_i/(Gz^\star - h)_i$ for $i \not\in \mathcal{I}$ and then solve the nonsingular system

$$\begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & 0 & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda_{\mathcal{I}} \\ \nu \end{bmatrix} = \begin{bmatrix} a - G_{\bar{\mathcal{I}}}^T \lambda_{\bar{\mathcal{I}}} \\ b_{\mathcal{I}} \\ c. \end{bmatrix} \tag{21}$$

Alternatively, suppose that we have both $\lambda_i^\star = 0$ and $(Gz^\star - h)_i = 0$. Then although the KKT matrix is now singular (any row for which $\lambda_i^\star = 0$ and $(Gz^\star - h)_i = 0$ will be all zero), there still exists a solution to the system (11), because the right hand side is always in the range of $D(\lambda^\star)$ and so will also be zero for these rows. In this case there will no longer be a *unique* solution, corresponding to the subdifferentiable but not differentiable case. □

The next two results show the representational power of the OptNet layer, specifically how an OptNet layer compares to the common linear layer followed by a ReLU activation. The first theorem shows that an OptNet layer can approximate arbitrary elementwise piecewise-linear functions, and so among other things can represent a ReLU layer.

Theorem 2. *Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be an elementwise piecewise linear function with k linear regions. Then the function can be represented as an OptNet layer using $O(nk)$ parameters. Additionally, the layer $z_{i+1} = \max Wz_i + b, 0$ for $W \in \mathbb{R}^{n \times m}, b \in \mathbb{R}^n$ can be represented by an OptNet layer with $O(mn)$ parameters.*

Proof. The proof that an OptNet layer can represent any piecewise linear univariate function relies on the fact that we can represent any such function in "sum-of-max" form

$$f(x) = \sum_{i=1}^{k} w_i \max\{a_i x + b, 0\} \tag{22}$$

where $w_i \in \{-1, 1\}$, $a_i, b_i \in \mathbb{R}$ (to do so, simply proceed left to right along the breakpoints of the function adding a properly scaled linear term to fit the next piecewise section). The OptNet layer simply represents this function directly.

That is, we encode the optimization problem

$$\begin{aligned} \underset{z \in \mathbb{R}, t \in \mathbb{R}^k}{\text{minimize}} \;\; & \|t\|_2^2 + (z - w^T t)^2 \\ \text{subject to} \;\; & a_i x + b_i \leq t_i, \quad i = 1, \ldots, k \end{aligned} \tag{23}$$

Clearly, the objective here is minimized when $z = w^T t$, and t is as small as possible, meaning each t must either be at its bound $a_i x + b \leq t_i$ or, if $a_i x + b < 0$, then $t_i = 0$ will be the optimal solution due to the objective function. To obtain a multivariate but elementwise function, we simply apply this function to each coordinate of the input x.

To see the specific case of a ReLU network, note that the layer

$$z = \max\{Wx + b, 0\} \tag{24}$$

is simply equivalent to the OptNet problem

$$\begin{aligned} \underset{z}{\text{minimize}} \;\; & \|z - Wx - b\|_2^2 \\ \text{subject to} \;\; & z \geq 0. \end{aligned} \tag{25}$$

□

Finally, we show that the converse does not hold: that there are function representable by an OptNet layer which cannot be represented exactly by a two-layer ReLU layer, which take exponentially many units to approximate (known to be a universal function approximator). A simple example of such a layer (and one which we use in the proof) is just the max over three linear functions $f(z) = \max\{a_1^T x, a_2^T x, a_3^T x\}$.

Theorem 3. *Let $f(z) : \mathbb{R}^n \to \mathbb{R}$ be a scalar-valued function specified by an OptNet layer with p parameters. Conversely, let $f'(z) = \sum_{i=1}^{m} w_i \max a_i^T z + b_i, 0$ be the output of a two-layer ReLU network. Then there exist functions that the ReLU network cannot represent exactly over all of $\mathbb{R}$, and which require $O(c^p)$ parameters to approximate over a finite region.*

Proof. The final theorem simply states that a two-layer ReLU network (more specifically, a ReLU followed by a linear layer, which is sufficient to achieve a universal function approximator), can often require exponentially many more units to approximate a function specified by an OptNet layer. That is, we consider a single-output ReLU network, much like in the previous section, but defined for multi-variate inputs.

$$f(x) = \sum_{i=1}^{m} w_i \max\{a_i^T x + b, 0\} \tag{26}$$

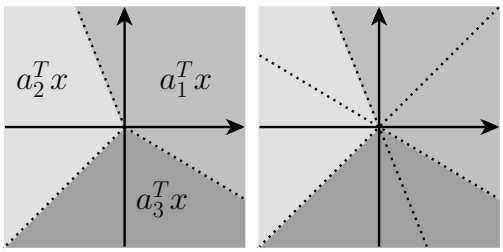


Figure 1: Creases for a three-term pointwise maximum (left), and a ReLU network (right).

Although there are many functions that such a network cannot represent, for illustration we consider a simple case of a maximum of three linear functions

$$f'(x) = \max\{a_1^T x, a_2^T x, a_3^T x\} \tag{27}$$

To see why a ReLU is not capable of representing this function exactly, even for $x \in \mathbb{R}^2$, note that any sum-of-max function, due to the nature of the term $\max a_i^T x + b_i, 0$ as stated above must have "creases" (breakpoints in the piecewise linear function), than span the entire input space; this is in contrast to the max terms, which can have creases that only partially span the space. This is illustrated in Figure 1. It is apparent, therefore, that the two-layer ReLU cannot exactly approximate the three maximum term (any ReLU network would necessarily have a crease going through one of the linear region of the original function). Yet this max function can be captured by a simple OptNet layer

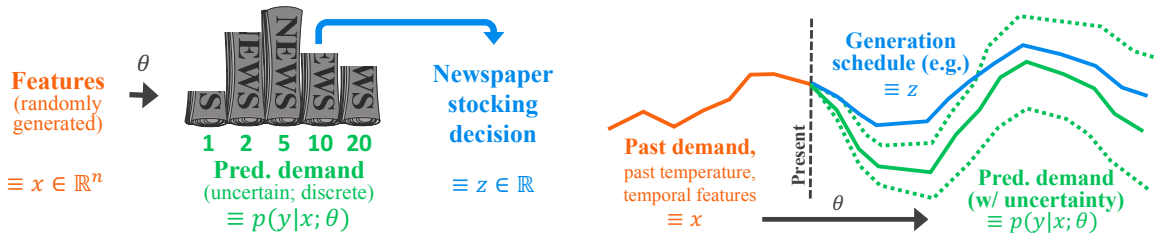
$$\begin{aligned} &\underset{z}{\text{minimize}} \;\; z^2 \\ &\text{subject to} \;\; a_i^T x \leq z, \; i = 1, \ldots, 3. \end{aligned} \tag{28}$$

The fact that the ReLU network is a universal function approximator means that the we *are* able to approximate the three-max term, but to do so means that we require a dense covering of points over the input space, choose an equal number of ReLU terms, then choose coefficients such that we approximate the underlying function on this points; however, for a large enough radius this will require an exponential size covering to approximate the underlying function arbitrarily closely. □

3.2.6 Limitations of the method

Although, as we will show shortly, the OptNet layer has several strong points, we also want to highlight the potential drawbacks of this approach. First, although, with an efficient batch solver, integrating an OptNet layer into existing deep learning architectures is potentially practical, we do note that solving optimization problems exactly as we do here has has cubic complexity in the number of variables and/or constraints. This contrasts with the quadratic complexity of standard feedforward layers. This means that we *are* ultimately limited to settings where the number of hidden variables in an OptNet layer is not too large (less than 1000 dimensions seems to be the limits of what we currently find to the be practical, and substantially less if one wants real-time results for an architecture).

Secondly, there are many improvements to the OptNet layers that are still possible. Our QP solver, for instance, uses fully dense matrix operations, which makes the solves very efficient for GPU solutions, and which also makes sense for our general setting where the coefficients of the quadratic problem can be learned. However, for setting many real-world optimization problems (and hence



(a) Inventory stock problem

(b) Load forecasting problem

Figure 2: Features x, model predictions y, and policy z for the two experiments.

for architectures that wish to more closely mimic some real-world optimization problem), there is often substantial structure (e.g., sparsity), in the data matrices that can be exploited for efficiency. There is of course no prohibition of incorporating sparse matrix methods into the fast custom solver, but doing so would require substantial added complexity, especially regarding efforts like finding minimum fill orderings for different sparsity patterns of the KKT systems. In our open source solver `qpth`, we have started experimenting with cuSOLVER's batched sparse QR factorizations and solves.

Lastly, we note that while the OptNet layers can be trained just as any neural network layer, since they are a new creation and since they have manifolds in the parameter space which have no effect on the resulting solution (e.g., scaling the rows of a constraint matrix and its right hand side does not change the optimization problem), there is admittedly more tuning required to get these to work. This situation is common when developing new neural network architectures and has also been reported in the similar architecture of [Schmidt and Roth, 2014]. Our hope is that techniques for overcoming some of the challenges in learning these layers will continue to be developed in future work.

4 Results and Discussion

Here we present results on both the end-to-end task learning and differentiable optimization settings.

4.1 Task-based model learning

We consider two applications of our task-based methodology, one to a synthetic inventory stock problem, and one to an energy scheduling task based on over eight years of real electrical grid data. In both cases, we demonstrate that the task-based end-to-end approach can substantially improve upon other alternatives. Source code for all experiments will be released with the full paper.

4.1.1 Inventory Stock Problem

Problem definition To highlight the performance of the algorithm in a setting where the true underlying model is known to us, we consider a "conditional" variation of the classical inventory stock problem [Shapiro and Philpott, 2007]. In this problem, a company must order some quantity z of a product to minimize costs over some stochastic demand y, whose distribution in turn is affected by some observed features x (Figure 2a). There are linear and quadratic costs on amount of product ordered, plus different linear/quadratic costs on overorders $[z - y]_+$ and underorders

$[y-z]_+$. The objective is given by:

$$f_{stock}(y,z) = c_0 z + \frac{1}{2}q_0 z^2 + c_b[y-z]_+ + \frac{1}{2}q_b([y-z]_+)^2 + c_h[z-y]_+ + \frac{1}{2}q_h([z-y]_+)^2, \tag{29}$$

where $[v]_+ = \max\{v, 0\}$. For a specific choice of probability model $p(y|x;\theta)$, our proxy stochastic programming problem can then be written as

$$\underset{z}{\text{minimize}} \quad \mathbf{E}_{y\sim p(y|x;\theta)}[f_{stock}(y,z)]. \tag{30}$$

To simplify the setting, we further assume that the demands are discrete, taking on values $d_1, \ldots, d_k$ with probabilities (conditional on x) $(p_\theta)_i \equiv p(y = d_i|x;\theta)$. Thus our stochastic programming problem (30) can be written succinctly as a joint quadratic program[5]

$$\begin{array}{ll} \underset{z\in\mathbb{R}, z_b, z_h \in \mathbb{R}^k}{\text{minimize}} & c_0 z + \frac{1}{2}q_0 z^2 + \sum_{i=1}^{k} (p_\theta)_i \left(c_b (z_b)_i + \frac{1}{2} q_b (z_b)_i^2 + c_h (z_h)_i + \frac{1}{2} q_h (z_h)_i^2 \right) \\ \text{subject to} & d - z\vec{1} \le z_b, \quad z\vec{1} - d \le z_h, \quad z, z_h, z_b \ge 0. \end{array} \tag{31}$$

Further details of this approach are given in the appendix.

Experimental setup We examine our algorithm under two main conditions: where the true model is linear, and where it is nonlinear. In all cases we generate problem instances by randomly sampling some $x \in \mathbb{R}^n$, and then generating $p(y|x;\theta)$ according to either $p(y|x;\theta) \propto \exp(\Theta^T x)$ (linear true model) or $p(y|x;\theta) \propto \exp((\Theta^T x)^2)$ (nonlinear true model) for some $\Theta \in \mathbb{R}^{n\times k}$. We compare the following approaches on these tasks: 1) The QP allocation based upon the true model (which gives optimal performance); 2) MLE approaches (with a linear or nonlinear probability model) that fit a model to the data, and then compute the allocation by solving the QP; 3) end-to-end neural network policy models (using linear or nonlinear hypotheses); and 4) our task-based learning models (with a linear or nonlinear probability model). In all cases we evaluate test performance by running on 1000 random examples, and evaluate performance over 10 folds of different true $\theta^\star$ parameters.

Figures 3(a) and (b) show the performance of these methods given a linear true model, with linear and nonlinear model hypotheses, respectively. As expected, the linear MLE approach performs best, as the true underlying model is in the class of distributions that it can represent and thus solving the stochastic programming problem is a very strong proxy for solving the true optimization problem under the real distribution. While the true model is also contained within the nonlinear MLE's generic nonlinear distribution class, we see that this method requires more data to converge, and in the meantime makes error tradeoffs that are ultimately not the correct tradeoffs for the task at hand; our task-based approach thus outperforms this approach. The task-based approach also substantially outperforms the policy-optimizing neural network, highlighting the fact that it is more data-efficient to run the learning process "through" a reasonable model. Note that here it does not make a difference whether we use the linear or nonlinear model in the task-based approach.

Figures 3(c) and (d) show performance in the case of a nonlinear true model, with linear and nonlinear model hypotheses, respectively. Case (c) represents the "non-realizable" case, where the true underlying distribution cannot be parameterized in the model hypothesis class. Here, the

[5]This is referred to as a two-stage stochastic programming problem (though a very trivial example of one), where first stage variables consist of the amount of product to buy before observing demand, and second-stage variables consist of how much to sell back or additionally purchase once the true demand has been revealed.

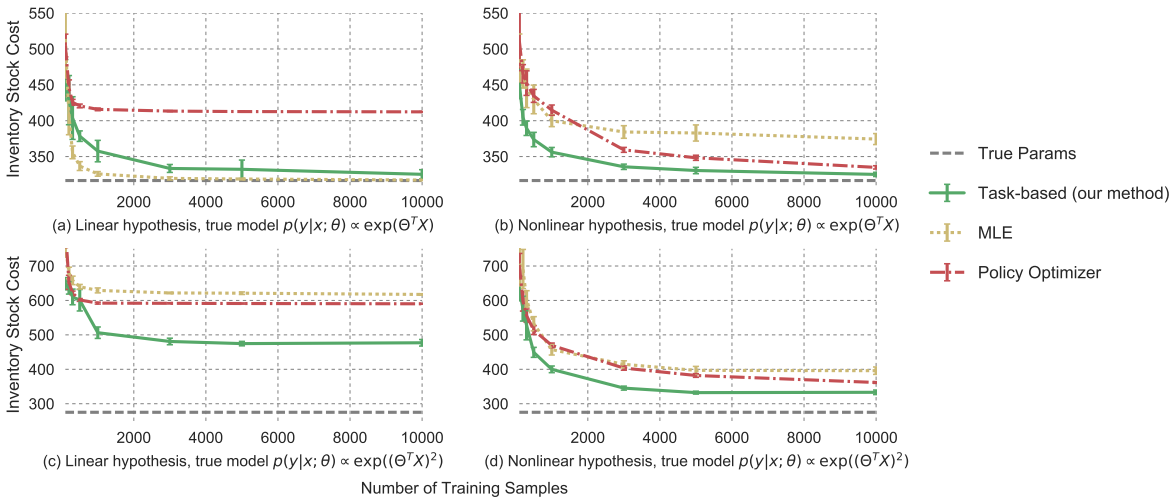


Figure 3: Inventory problem results for 10 runs over a representative instantiation of true parameters. Cost is evaluated over 1000 testing samples (lower is better). The linear MLE performs best for a true linear model. In all other cases, the task-based models outperform their MLE and policy counterparts.

linear MLE, as expected, performs very poorly: it cannot capture the true underlying distribution, and thus solving a stochastic program with this problem would not be expected to perform well. The linear policy model similarly performs poorly. Importantly, the task-based approach with the *linear* model performs much better here: despite the fact that it still has a misspecified model, the task-based nature of the learning process lets us learn a *different* linear model than the MLE version, which is particularly tuned to the distribution and loss of the task. Finally, also as to be expected, the non-linear models perform better than the linear models in this scenario, but again with the task-based non-linear model outperforming the nonlinear MLE and end-to-end policy approaches.

4.1.2 Load Forecasting and Generator Scheduling

We next consider a more realistic grid-scheduling task, based upon over eight years of real electrical grid data. In this setting, a power system operator must decide how much electricity generation $z \in \mathbb{R}^{24}$ to schedule for each hour in the next 24 hours based on some (unknown) distribution over electricity demand (Figure 2b). Given a particular realization y of demand, we impose penalties for both generation excess (γ_e) and generation shortage (γ_s), with $\gamma_s \gg \gamma_e$. We also add a quadratic regularization term, indicating a preference for generation schedules that closely match demand realizations. Finally, we impose a ramping constraint c_r restricting the change in generation between consecutive timepoints, reflecting physical limitations associated with quick changes in electricity output levels. These are reasonable proxies for the actual economic costs incurred by electrical grid operators when scheduling generation, and can be written as the stochastic programming problem

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \sum_{i=1}^{24} \mathbf{E}_{y\sim p(y|x;\theta)} \left[\gamma_s [y_i - z_i]_+ + \gamma_e [z_i - y_i]_+ + \frac{1}{2}(z_i - y_i)^2 \right] \\ \text{subject to} \quad & |z_i - z_{i-1}| \leq c_r \ \forall i, \end{aligned} \tag{32}$$

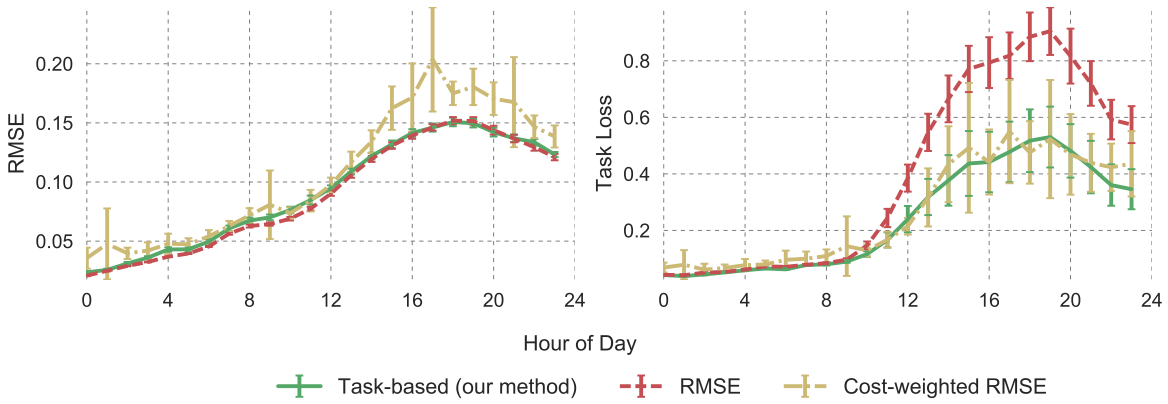


Figure 4: Results for 10 runs of the generation-scheduling problem. (Lower loss is better.) As expected, the RMSE net achieves the lowest RMSE for its predictions. However, the task net outperforms the RMSE net on task loss by 38.6%, and the cost-weighted RMSE by 8.6%.

where $[v]_+ = \max\{v, 0\}$. Assuming (as we will in our model), that y_i is a Gaussian random variable with mean μ_i and variance σ_i^2, then this expectation has a closed form that can be computed via analytically integrating the Gaussian pdf.[6] We then use sequential quadratic programming (SQP) to iteratively approximate the resultant convex objective as a quadratic objective, iterate until convergence, and then compute the necessary Jacobians using the quadratic approximation at the solution, which gives the correct Hessian and gradient terms. Details are given in the appendix.

To develop a predictive model, we make use of a highly-tuned load forecasting methodology. Specifically, we input the past day's electrical load and temperature, the next day's temperature forecast, and additional features such as non-linear functions of the temperatures, binary indicators of weekends or holidays, and yearly sinusoidal features. We then predict the electrical load over all 24 hours of the next day. We employ a 2-hidden layer neural network for this purpose, with an additional residual connection from the inputs to the outputs initialized to the linear regression solution. An illustration of the architecture is shown in Figure 3. We train the model to minimize the mean squared error between its predictions and the actual load (giving the mean prediction μ_i), and compute σ_i^2 as the (constant) empirical variance between the predicted and actual values. In all cases we use 7 years of data to train the model, and 1.75 subsequent years for testing.

Using the (mean and variance) predictions of this base model, we solve the generator scheduling problem by solving the optimization problem (32), learning network parameters by minimizing the task loss. We compare against a traditional stochastic programming model that minimizes just the RMSE, as well as a task-cost-weighted RMSE that periodically reweights training samples given their task loss. (A pure end-to-end network is not shown, as it was not able to sufficiently learn the ramp constraints and thus generated infeasible schedules. We could not obtain good performance even ignoring this infeasibility.)

Figure 4 shows the performance of the three models. As expected, the RMSE model performs best when measured by the RMSE of its predictions (its objective). However, the task-based model substantially outperforms the RMSE model when evaluated on task loss, the actual objective that

[6] Part of the philosophy behind applying our approach here is that we *know* the Gaussian assumption is incorrect: the true underlying load is neither Gaussian distributed nor homoskedastic. However, these assumptions are exceedingly common in practice, as they enable easy model learning and exact analytical solutions. Thus, training the (still Gaussian) system with a task-based loss retains computational tractability while still allowing us to modify the distributional parameters to improve actual performance on the task at hand.

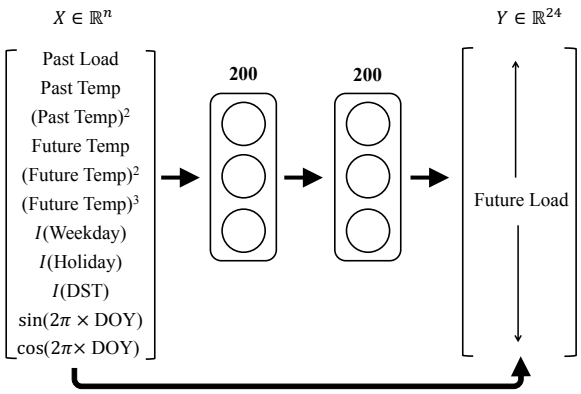


Figure 5: 2-hidden layer neural network to predict hourly electrical load for the next day.

the system operator cares about: in relative terms, we improve upon the performance of the traditional stochastic programming methodology by 38.6%. The cost-weighted RMSE's performance is extremely variable, as it is not robust to the exact timing of reweighting. It is also worth noting that a cost-weighted RMSE approach is only possible when direct costs can be assigned independently to each decision point, i.e. when costs do not depend on multiple decision points (as in this experiment); our task-based method, however, accommodates the (typical) more general setting. Overall, the task net improves upon the cost-weighted RMSE by 8.6%.

4.2 Differentiable optimization

In this section, we present several experimental results that highlight the capabilities of the QP OptNet layer. Specifically we look at 1) computational efficiency over exiting solvers; 2) the ability to improve upon existing convex problems such as those used in signal denoising; 3) integrating the architecture into an generic deep learning architectures; and 3) performance of our approach on a problem that is very challenging for current approaches. performance of our approach can sometimes vastly improve upon existing deep architectures architectures. In particular, we want to emphasize the results of our system on learning the game of (4x4) Sudoku, a well-known logical puzzle; to the best of our knowledge, ours in the first type of end-to-end differentiable architecture that can learn problems such as this one based solely upon examples with no a priori knowledge of the rules of Sudoku. The code and data for our experiments are open sourced at `https://github.com/locuslab/optnet` and our batched QP solver is available as a library at `https://github.com/locuslab/qpth`.

4.2.1 Batch QP solver performance

Our OptNet layers are much more computationally expensive than a linear or convolutional layer and a natural question is to ask what the performance difference is. We set up an experiment comparing a linear layer to a QP OptNet layer with a mini-batch size of 128 on CUDA with randomly generated input vectors sized 10, 100, and 1000. Figure 4 shows timing results of the forward and backward passes.

Our next experiment illustrates why standard baseline QP solvers like CPLEX and Gurobi without batch support are too computationally expensive for QP OptNet layers to be tractable. We run our solver on an unloaded Titan X GPU and Gurobi on an unloaded quad-core Intel Core i7-5960X CPU @ 3.00GHz. We set up the same random QP of the form (6) across all three

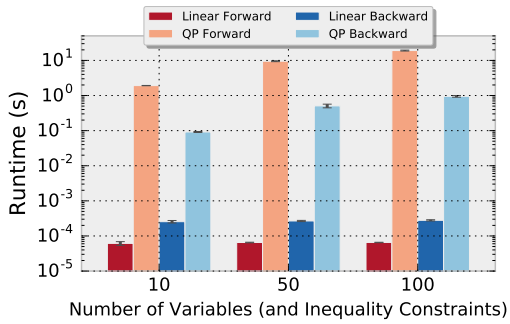


Figure 6: Performance of a linear layer and a QP layer.

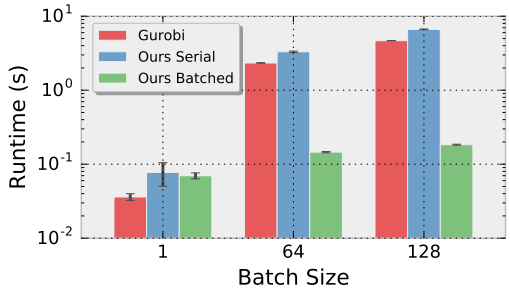


Figure 7: Performance of Gurobi and our QP solver.

frameworks and vary the number of variable, constraints, and batch size.[7]

Figure 5 shows the means and standard deviations of running each trial 10 times, showing that our solver outperforms Gurobi, itself a highly tuned solver, in all batched instances. For the minibatch size of 128, we solve all problems in an average of 0.18 seconds, whereas Gurobi tasks an average of 4.7 seconds. In the context of training a deep architecture this type of speed difference for a single minibatch can make the difference between a practical and a completely unusable solution.

4.2.2 Total variation denoising

Our next experiment studies how we can use the OptNet architecture to *improve* upon signal processing techniques that currently use convex optimization as a basis. Specifically, our goal in this case is to denoise a noisy 1D signal given training data consistency of noisy and clean signals generated from the same distribution. Such problems are often addressed by convex optimization procedures, and (1D) total variation denoising is a particularly common and simple approach. Specifically, the total variation denoising approach attempts to smooth some noisy observed signal y by solving the optimization problem

$$\underset{z}{\text{argmin}} \frac{1}{2}||y-z|| + \lambda||Dz||_1 \tag{33}$$

where D is the first-order differencing operation, which can be expressed in matrix form by $D_i = e_i - e_{i+1}$ Penalizing the ℓ_1 norm of the signal *difference* encourages this difference to be sparse, i.e.,

[7]Experimental details: we sample entries of a matrix U from a random uniform distribution and set $Q = U^T U + 10^{-3} I$, sample G with random normal entries, and set h by selecting generating some z_0 random normal and s_0 random uniform and setting $h = Gz_0 + s_0$ (we didn't include equality constraints just for simplicity, and since the number of inequality constraints in the primary driver of complexity for the iterations in a primal-dual interior point method). The choice of h guarantees the problem is feasible.

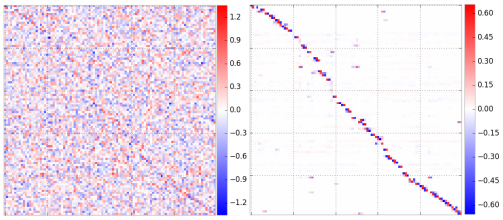


Figure 8: Initial and learned difference operators for denoising.

the number of changepoints of the signal is small, and we end up approximating y by a (roughly) piecewise constant function.

To test this approach and competing ones on a denoising task, we generate piecewise constant signals (which are the desired outputs of the learning algorithm) and corrupt them with independent Gaussian noise (which form the inputs to the learning algorithm). The summary training and testing performance of the four approaches we describe are shown in Table 1.

Baseline: Total variation denoising To establish a baseline for denoising performance with total variation, we run the above optimization problem varying values of λ between 0 and 100. The procedure performs best with a choice of $\lambda \approx 13$, and achieves a minimum test MSE on our task of about 16.5 (the units here are unimportant, the only relevant quantity is the relative performances of the different algorithms).

Baseline: Learning with a fully-connected neural network An alternative approach to denoising is by learning from data. A function $f_\theta(x)$ parameterized by θ can be used to predict the original signal. The optimal θ can be learned by using the mean squared error between the true and predicted signals. Denoising is typically a difficult function to learn and Table 1 shows that a fully-connected neural network perform substantially worse on this denoising task than the convex optimization problem. Section **??** shows the convergence of the fully-connected network.

Learning the differencing operator with OptNet Between the feedforward neural network approach and the convex total variation optimization, we could instead use a generic OptNet layers that effectively allowed us to solve (33) using *any* denoising matrix, which we randomly initialize. While the accuracy here is substantially lower than even the fully connected case, this is largely the result of learning an over-regularized solution to D. This is indeed a point that should be addressed in future work (we refer back to our comments in the previous section on the potential challenges of training these layers), but the point we want to highlight here is that the OptNet layer seems to be learning something very interpretable and understandable. Specifically, Figure 6 shows the D matrix of our solution before and after learning (we permute the rows to make them ordered by the magnitude of where the large-absolute-value entries occurs). What is interesting in this picture is that the learned D matrix typically captures exactly the same intuition as the D matrix used by total variation denoising: a mainly sparse matrix with a few entries of alternating sign next to each other. This implies that for the data set we have, total variation denoising is indeed the "right" way to think about denoising the resulting signal, but if some other noise process were to generate the data, then we can learn that process instead. We can then attain lower actual error for the method (in this case similar though slightly higher than the TV solution), by fixing the learned sparsity of the D matrix and then fine tuning.

Method	Train MSE	Test MSE
FC Net	18.5	29.8
Pure OptNet	52.9	53.3
Total Variation	16.3	16.5
OptNet Tuned TV	13.8	**14.4**

Table 1: Denoising task accuracies.

			3
1			
		4	
4			1

2	4	1	3
1	3	2	4
3	1	4	2
4	2	3	1

Figure 9: Example 4x4 Sudoku puzzle, showing initial problem and solution.

Fine-tuning and improving the total variation solution To finally highlight the ability of the OptNet methods to *improve* upon the results of a convex program, specifically tailoring to the data. Here, we use the same OptNet architecture as in the previous subsection, but initialize D to be the differencing matrix as in the total variation solution. As shown in Table 1, the procedure is able to improve both the training and testing MSE over the TV solution, specifically improving upon test MSE by 12%.

4.2.3 Sudoku

Finally, we present the main illustrative example of the representational power of our approach, the task of learning the game of Sudoku. Sudoku is a popular logical puzzle, where a (typically 9x9) grid of points must be arranged given some initial point, so that each row, each column, and each 3x3 grid of points must contain one of each number 1 through 9. We consider the slightly simpler case of 4x4 Sudoku puzzles, with numbers 1 through 4, as shown in Figure **??**.

Sudoku is fundamentally a constraint satisfaction problem, and is trivial for computers to solve when told the rules of the game. However, if we do not know the rules of the game, but are only presented with examples of unsolved and the corresponding solved puzzle, this is a challenging task. We consider this to be an interesting benchmark task for algorithms that seek to capture complex strict relationships between all input and output variables. The input to the algorithm consists of a 4x4 grid (really a 4x4x4 tensor with a one-hot encoding for known entries an all zeros for unknown entries), and the desired output is a 4x4x4 tensor of the one-hot encoding of the solution.

This is a problem where traditional neural networks have difficulties learning the necessary hard constraints. As a baseline inspired by the models at `https://github.com/Kyubyong/sudoku`, we implemented a multilayer feedforward network to attempt to solve Sudoku problems. Specifically, we report results for a network that has 10 convolutional layers with 512 3x3 filters each, and tried other architectures as well. The OptNet layer we use on this task is a completely generic QP in "standard form" with only positivity inequality constraints but an arbitrary constraint matrix $Ax = b$, a small $Q = 0.1I$ to make sure the problem is strictly feasible, and with the linear term q simply being the input one-hot encoding of the Sudoku problem. We know that Sudoku *can* be approximated well with a linear program (indeed, integer programming is a typical solution method for such problems), but the model here is told nothing about the rules of Sudoku.

We trained these models using ADAM [Kingma and Ba, 2014] to minimize the MSE (which we

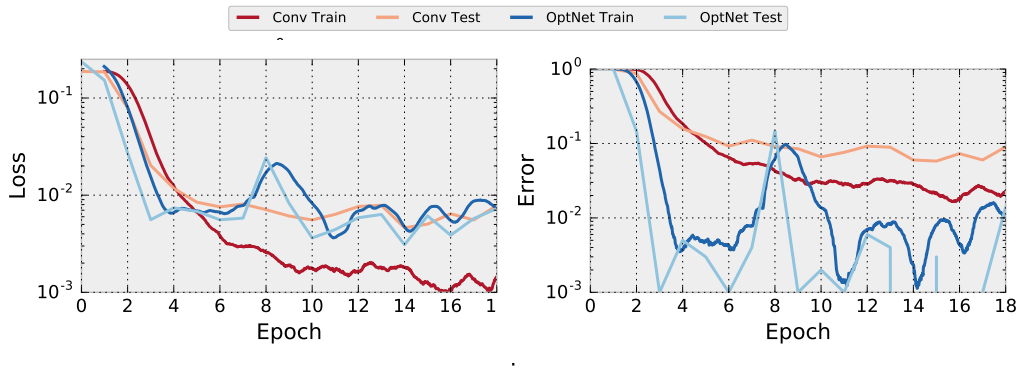


Figure 10: Sudoku training plots.

refer to as "loss") on a dataset we created consisting of 9000 training puzzles, and we then tested the models on 1000 different held-out puzzles. The error rate is the percentage of puzzles solved correctly if the cells are assigned to whichever index is largest in the prediction. Figure 8 shows that the convolutional is able to learn all of the necessary logic for the task and ends up over-fitting to the training data. We contrast this with the performance of the OptNet network, which learns most of the correct hard constraints within the first three epochs and is able to generalize much better to unseen examples.

5 Conclusions

This report has detailed a methodology for learning machine learning models that are intended to be used in the loop of a more complex decision-making process. The basic approach is to adjust the model parameters directly to optimize the closed-loop performance of the system, in addition to simply the predictive accuracy of the model. Doing so requires that we differentiate through optimization layers, and we provide a methodology and algorithmic implementation for doing so.

There are a number of potential directions of research that this work suggests. At a high level, the methodology suggests an alternative to classical model learning, where we incorporate task-driven objectives that can be mixed with classical learning objectives. However, the nature of the optimization itself, which we consider to be stochastic optimization based upon sequential quadratic programming and stochastic optimization, can be extended to additional approaches. And the question of determining the best model class for different scenarios, and efficiently solving the optimization problem for large-scale systems, still remains a challenging open problem for future work.

References

Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

Brandon Amos, Lei Xu, and J Zico Kolter. Input convex neural networks. *arXiv preprint arXiv:1609.07152*, 2016.

Somil Bansal, Roberto Calandra, Ted Xiao, Sergey Levine, and Claire J Tomlin. Goal-driven dynamics learning via bayesian optimization. *arXiv preprint arXiv:1703.09260*, 2017.

David Belanger and Andrew McCallum. Structured prediction energy networks. In *Proceedings of the International Conference on Machine Learning*, 2016.

David Belanger, Bishan Yang, and Andrew McCallum. End-to-end learning for structured prediction energy networks. *arXiv preprint arXiv:1703.05667*, 2017.

Yoshua Bengio. Using a financial training criterion rather than a prediction criterion. *International Journal of Neural Systems*, 8(04):433–443, 1997.

Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II: Approximate Dynamic Programming*. Athena Scientific, Belmont, MA, 4 edition, 2012.

J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Springer, New York, 2nd edition, 2011.

J Frédéric Bonnans and Alexander Shapiro. *Perturbation analysis of optimization problems*. Springer Science & Business Media, 2013.

Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

Philémon Brakel, Dirk Stroobandt, and Benjamin Schrauwen. Training energy-based models for time-series imputation. *Journal of Machine Learning Research*, 14(1):2771–2797, 2013.

John A Buzacott and J George Shanthikumar. *Stochastic models of manufacturing systems*, volume 4. Prentice Hall Englewood Cliffs, NJ, 1993.

E.F. Camacho and C. Bordons. *Model Predictive Control*. Springer, London, 2003.

Liang-Chieh Chen, Alexander G Schwing, Alan L Yuille, and Raquel Urtasun. Learning deep structured models. In *Proceedings of the International Conference on Machine Learning*, 2015.

Frank H Clarke. Generalized gradients and applications. *Transactions of the American Mathematical Society*, 205:247–262, 1975.

Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A Survey on Policy Search for Robotics. *Foundations and Trends in Machine Learning*, 2(1-2):1–142, 2011. doi: 10.1561/2300000021.

Justin Domke. Generic methods for optimization-based modeling. In *AISTATS*, volume 22, pages 318–326, 2012.

Asen L Dontchev and R Tyrrell Rockafellar. Implicit functions and solution mappings. *Springer Monogr. Math.*, 2009.

Anthony V Fiacco and Yo Ishizuka. Sensitivity and stability analysis for nonlinear programming. *Annals of Operations Research*, 27(1):215–235, 1990.

Ian Goodfellow, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Multi-prediction deep boltzmann machines. In *Advances in Neural Information Processing Systems*, pages 548–556, 2013.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

Stephen Gould, Basura Fernando, Anoop Cherian, Peter Anderson, Rodrigo Santa Cruz, and Edison Guo. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. *arXiv preprint arXiv:1607.05447*, 2016.

Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *ICML*, volume 14, pages 1764–1772, 2014.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Ken Harada, Jun Sakuma, and Shigenobu Kobayashi. Local search for multiobjective function optimization: pareto descent method. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 659–666. ACM, 2006.

Tamir Hazan, Joseph Keshet, and David A McAllester. Direct loss minimization for structured prediction. In *Advances in Neural Information Processing Systems*, pages 1594–1602, 2010.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

Matthew Johnson, David K Duvenaud, Alex Wiltschko, Ryan P Adams, and Sandeep R Datta. Composing graphical models with neural networks for structured representations and fast inference. In *Advances in Neural Information Processing Systems*, pages 2946–2954, 2016.

Michael Peter Kennedy and Leon O Chua. Neural networks for nonlinear programming. *IEEE Transactions on Circuits and Systems*, 35(5):554–562, 1988.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Karl Kunisch and Thomas Pock. A bilevel optimization approach for parameter learning in variational models. *SIAM Journal on Imaging Sciences*, 6(2):938–983, 2013.

Yann LeCun, Urs Muller, Jan Ben, Eric Cosatto, and Beat Flepp. Off-road obstacle avoidance through end-to-end learning. In *NIPS*, pages 739–746, 2005.

Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and F Huang. A tutorial on energy-based learning. *Predicting structured data*, 1:0, 2006.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

Walter E Lillo, Mei Heng Loh, Stefen Hui, and Stanislaw H Zak. On solving constrained optimization problems with neural networks: A penalty method approach. *IEEE Transactions on neural networks*, 4(6):931–940, 1993.

Jeff Linderoth, Alexander Shapiro, and Stephen Wright. The empirical behavior of sampling methods for stochastic programming. *Annals of Operations Research*, 142(1):215–241, 2006.

Per Lötstedt. Numerical simulation of time-dependent contact and friction problems in rigid body mechanics. *SIAM journal on scientific and statistical computing*, 5(2):370–393, 1984.

X Magnus and Heinz Neudecker. Matrix differential calculus. *New York*, 1988.

Julien Mairal, Francis Bach, and Jean Ponce. Task-driven dictionary learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4):791–804, 2012.

Shie Mannor, R Rubinstein, and Y Gat. The cross entropy method for fast policy search. In *Proceedings of the 20th International Conference on Machine Learning*, 2003.

Jacob Mattingley and Stephen Boyd. Cvxgen: A code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, 2012.

Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. *arXiv preprint arXiv:1611.02163*, 2016.

Manfred Morari and Jay H Lee. Model predictive control: past, present and future. *Computers & Chemical Engineering*, 23(4):667–682, 1999.

Hossam Mossalam, Yannis M Assael, Diederik M Roijers, and Shimon Whiteson. Multi-objective deep reinforcement learning. *arXiv preprint arXiv:1610.02707*, 2016.

Jian Peng, Liefeng Bo, and Jinbo Xu. Conditional neural fields. In *Advances in neural information processing systems*, pages 1419–1427, 2009.

Warren B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality.* John Wiley & Sons, Hoboken, NJ, 2 edition, 2011.

Warren B. Powell. Clearing the Jungle of Stochastic Optimization. *Informs TutORials in Operations Research 2014*, (October), 2014. doi: http://dx.doi.org/10.1287/educ.2014.0128.

Warren B. Powell. A Unified Framework for Optimization under Uncertainty. *Informs TutORials in Operations Research*, 2016.

Martin L. Puterman. *Markov Decision Processes.* John Wiley and Sons, 2nd edition, 2005.

R Tyrrell Rockafellar and Roger J-B Wets. Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of operations research*, 16(1):119–147, 1991.

Shankar Sastry and Marc Bodson. *Adaptive control: stability, convergence and robustness.* Courier Corporation, 2011.

Uwe Schmidt and Stefan Roth. Shrinkage fields for effective image restoration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2774–2781, 2014.

Alexander Shapiro and Andy Philpott. A tutorial on stochastic programming. *Manuscript. Available at www2. isye. gatech. edu/ ashapiro/ publications. html* , 17, 2007.

Yang Song, Alexander G Schwing, Richard S Zemel, and Raquel Urtasun. Training deep neural networks via direct loss minimization. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 2169–2177, 2016.

James C Spall. *Introduction to Stochastic Search and Optimization: Estimation, simulation and control.* John Wiley & Sons, Hoboken, NJ, 2003.

Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

Robert F. Stengel. *Stochastic optimal control: theory and application*. John Wiley & Sons, Hoboken, NJ, 1986.

Veselin Stoyanov, Alexander Ropson, and Jason Eisner. Empirical risk minimization of graphical model parameters given approximate inference, decoding, and model structure. In *AISTATS*, pages 725–733, 2011.

Richard S. Sutton and A Barto. *Reinforcement Learning*, volume 35. MIT Press, Cambridge, MA, 1998.

Csaba Szepesvári. *Algorithms for Reinforcement Learning*, volume 4. Morgan and Claypool, jan 2010. doi: 10.2200/S00268ED1V01Y201005AIM009.

Aviv Tamar, Sergey Levine, Pieter Abbeel, YI WU, and Garrett Thomas. Value iteration networks. In *Advances in Neural Information Processing Systems*, pages 2146–2154, 2016.

Marshall F Tappen, Ce Liu, Edward H Adelson, and William T Freeman. Learning gaussian conditional random fields for low-level vision. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

Ryan W Thomas, Daniel H Friend, Luiz A Dasilva, and Allen B Mackenzie. Cognitive networks: adaptation and learning to achieve end-to-end performance objectives. *IEEE Communications Magazine*, 44(12):51–57, 2006.

Kristof Van Moffaert and Ann Nowé. Multi-objective reinforcement learning using sets of pareto dominating policies. *Journal of Machine Learning Research*, 15(1):3483–3512, 2014.

Stein W Wallace and Stein-Erik Fleten. Stochastic programming models in energy. *Handbooks in operations research and management science*, 10:637–677, 2003.

Kai Wang, Boris Babenko, and Serge Belongie. End-to-end scene text recognition. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1457–1464. IEEE, 2011.

Tao Wang, David J Wu, Adam Coates, and Andrew Y Ng. End-to-end text recognition with convolutional neural networks. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 3304–3308. IEEE, 2012.

Marco A Wiering, Maikel Withagen, and Mădălina M Drugan. Model-based multi-objective reinforcement learning. In *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2014 IEEE Symposium on*, pages 1–6. IEEE, 2014.

Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1529–1537, 2015.

William T Ziemba and Raymond G Vickson. *Stochastic optimization models in finance*, volume 1. World Scientific, 2006.

List of Symbols, Abbreviations, and Acronyms

- **ADAM** - adaptive gradient descent method
- **CRF** - conditional random field
- **GPU** - graphics processing unit
- **KKT condition** - Karush-Kuhn-Tucker conditions (named after authors)
- **ML** - machine learning
- **MLE** - maximum likelihood estimate
- **PDIPM** - primal dual interior point method
- **QP** - quadratic program
- **ReLU** - rectified linear unit
- **RMSE** - root mean squared error